ETH Zürich
Chair of Software Engineering

Algebra Unifies Operational Calculi

Stephan van Staden, ETH Zurich, Switzerland
Tony Hoare, Microsoft Research Cambridge, UK

Introduction

- Operational calculi facilitate reasoning about program execution
- Various flavours, e.g.
  - small/big step
  - with/without explicit store
  - with explicit evaluation contexts, …
- Aim of this work: to unify various calculi by deriving their rules from algebraic laws
- The laws capture the essence of how programs behave and interact
- The calculi will capture/emphasize different aspects of the laws. Commonalities and differences become clear

Overview

- Operators & laws of programming
  - Including concurrency!
- General operational calculus
  - Judgement is a quintuple
  - Derive its rules by algebraic reasoning
- Specialise to familiar operational calculi (of Plotkin, Milner, Kahn, …)
- Conclusion

Algebra: carrier set and operators

Carrier set: D (for descriptions)

- Think: sets of executions (behaviours)
  Can describe programs, actions, states, etc.
- Ranged over by (P, Q, R, S, s)

Operators: intuition and some basic laws

- (D, $\bigvee$, $\bigwedge$) is a lattice
- $\bigvee$ is nondeterministic choice
- Refinement: $P \leq Q \equiv P \bigvee Q = Q$ (partial order)
  P has the same or fewer behaviours compared to Q
- *skip* ; * ‖

More laws of programming

(D, ;, *skip*) is a monoid

(D, ‖, *skip*) is a commutative monoid

All the binary operators distribute through $\bigvee$

- hence monotone

* has the algebraic properties of Kleene star

- monotone
- *skip* $\bigvee$ (P;P*) ≤ P*, … (see Kleene algebra)

Exchange law

$(P \| Q);(R \| S) \leq (P;R) \| (Q;S)$

- True of language interleaving, pomsets

Consequences:

$P;(Q \| R) \leq (P;Q) \| R$

$(P \| Q);R \leq P \| (Q;R)$

$P;Q \leq P \| Q$

Other examples of exchange: if $\bullet$ is monotone, then

$(P /\backslash Q) \bullet (R /\backslash S) \leq (P \bullet R) /\backslash (Q \bullet S)$

General operational calculus

$\langle P, s\rangle \text{ -}Q\text{-> } \langle P', s'\rangle \quad \equiv \quad Q \in \text{Actions} \quad \& \quad P \geq Q;P' \quad \& \quad s;Q \geq s'$

Actions contains descriptions of primitive actions

- executed in a single step
- defines the step size of a calculus (big-step: Actions ≡ D)
- Requirement: *skip* ∈ Actions

$P \geq Q;P'$: One possible way of executing P is to execute Q first and then P'

$s;Q \geq s'$: If s describes [illegible] Q, then s' describes a possible state [illegible]

How an action relates before/after programs

Separation of concerns!

How an action relates before/after states

General operational calculus

Derive rules of the calculus as theorems

Small-step concurrency: $\langle P, s\rangle \text{ -}Q\text{-> } \langle P', s'\rangle \Rightarrow \langle P \| R, s\rangle \text{ -}Q\text{-> } \langle P' \| R, s'\rangle$

Proof: Assume $Q \in \text{Actions}$ & $P \geq Q;P'$ & $s;Q \geq s'$

So $P \| R$

$\geq$ { $\|$ monotone, assumption $P \geq Q;P'$ }

$(Q;P') \| R$

$\geq$ { exchange law corollary: "$P;(Q \| R) \leq (P;Q) \| R$" }

$Q;(P' \| R)$

Corollary: $\langle P, s\rangle \text{ -}Q\text{-> } \langle \textit{skip}, s'\rangle \Rightarrow \langle P \| R, s\rangle \text{ -}Q\text{-> } \langle R, s'\rangle$

Big-step sequential composition. Provided $Q;Q' \in \text{Actions}$:

$\langle P, s\rangle \text{ -}Q\text{-> } \langle \textit{skip}, s'\rangle$ & $\langle P', s'\rangle \text{ -}Q'\text{-> } \langle \textit{skip}, s''\rangle$

$\Rightarrow \langle P;P', s\rangle \text{ -}Q;Q'\text{-> } \langle \textit{skip}, s''\rangle$

Plotkin calculus (SOS)

<P, s> --> <P', s'> ≡ ∃Q. <P, s> -Q-> <P', s'>
<P, s> --> s' ≡ ∃Q. <P, s> -Q-> <*skip*, s'>

Rules:
<*skip*, s> --> s
- From <*skip*, s> -*skip*-> <*skip*, s>

<P⋁P', s> --> <P, s>
- From <P⋁P', s> -*skip*-> <P, s>

<P, s> --> s' => <P‖R, s> --> <R, s'>
- From <P, s> -Q-> <*skip*, s'> => <P‖R, s> -Q-> <R, s'>

<P, s> --> <P', s'> => <P‖R, s> --> <P'‖R, s'>
- From <P, s> -Q-> <P', s'> => <P‖R, s> -Q-> <P'‖R, s'>

Milner calculus

Used in process calculi (e.g. CCS). Hides states:

P -Q-> P' ≡ ∀s. ∃s'. <P, s> -Q-> <P', s'>
(<=> Q ∈ Actions & P ≥ Q;P' and also <=> ∃s . ∃s'. <P, s> -Q-> <P', s'>)

P ∈ Actions => P -P-> *skip*

- Prefixing of CCS: P ∈ Actions => P;P' -P-> P'

P⋁P' -*skip*-> P (also written P⋁P' --> P . Note that --> = ≥)

- From <P⋁P', s> -*skip*-> <P, s>

P -Q-> *skip* => P‖R -Q-> R

- From <P, s> -Q-> <*skip*, s'> => <P‖R, s> -Q-> <R, s'>

P -Q-> P' => P‖R -Q-> P'‖R

- From <P, s> -Q-> <P', s'> => <P‖R, s> -Q-> <P'‖R, s'>

Reduction semantics with evaluation contexts

Two judgements: <P, s> --> <P', s'> and P --> P'

A *program context* is any function *pc* : D -> D such that:

P -Q-> P' => *pc*(P) -Q-> *pc*(P')

Idea: component P of *pc*(P) may be executed next

Examples (theorems): id λx. (x;P) λx. (x‖P) λx. (P‖x)

Theorem: Program contexts are closed under composition

Another example: λx. ((P‖(x‖P'));P'')

Reduction semantics with evaluation contexts

Only two rules with premises:

P --> P’ => *pc*(P) --> *pc*(P’)

P --> P’ => <*pc*(P), s> --> <*pc*(P’), s>

Examples using P\/P’ --> P and λx. (x‖P’’):

- (P\/P’)‖P’’ --> P‖P’’
- <(P\/P’)‖P’’, s> --> <P‖P’’, s>

For every non-*skip* action Q with <Q, s> --> s’ (Plotkin!), the calculus includes the (premise-free) rule:
<*pc*(Q), s> --> <*pc*(*skip*), s’>

skip is reduced by *skip*;P --> P *skip*‖P --> P P‖*skip* --> P

Kahn calculus (natural semantics)

Big steps: the set Actions contains every description

<P, s> --> s’ ≡ ∃Q. <P, s> -Q-> <*skip*, s’> (<=> s;P ≥ s’)

Actions are hidden and programs are executed to completion

<P, s> --> s’ & <P’, s’> --> s’’ => <P;P’, s> --> s’’

- From <P, s> -Q-> <*skip*, s’> & <P’, s’> -Q’-> <*skip*, s’’> => <P;P’, s> -Q;Q’-> <*skip*, s’’>

<P*, s> --> s

- From <P*, s> -*skip*-> <*skip*, s>

<P, s> --> s’ & <P*, s’> --> s’’ => <P*, s> --> s’’

- From <P, s> -Q-> <*skip*, s’> & <P*, s’> -Q’-> <*skip*, s’’> => <P*, s> -Q;Q’-> <*skip*, s’’>

Conclusion

Algebra unifies operational calculi

- Algebra can describe and explain a range of calculi (including ones not mentioned in this talk)
- Simpler and stronger than the calculi
- Facilitates a comparison of the calculi (differences and commonalities)

Why algebra?

- simple and elegant
- abstract:
  Specify only the properties of interest
  No commitment to a particular model
- extensible (incremental and modular, reusable)
- proved useful for unification in mathematics (e.g. arithmetic)